Exceptionally Fast non-linear 3D Image Registration using GPUs

Richard E. Ansorge, *Member, IEEE,* Steve J. Sawiak, and Guy B. Williams

***Abstract*–We have developed code to test the potential of GPUs for speeding up typical 3D Medical Image registration tasks. We have used NVIDIA's CUDA programming language because this is very close to C and hence such code is likely to have lasting value. We find registration using both affine and B-Spline based non-linear transformations can be greatly speeded up. A typical very high resolution registration using up to one B-Spline control point every two voxels can be completed in about 5 minutes using a Normalized Correlation cost function and in about eight minutes using a Normalized Mutual Information cost function.**

I. Introduction

DEDICATED PC graphics processor cards (GPUs) have a long history [1] and there is a parallel history of attempts to exploit these cards for medical imaging (and numerous other) applications [2], [3]. However until very recently these efforts have had little impact due to the difficulty of programming GPU cards and their rapid obsolescence. We believe this situation has now dramatically changed; in particular the latest generation of NVIDIA GPUs has a C like programming language CUDA [4] which offers a stable and easy to program platform capable of delivering exceptional performance. Many C++ extensions, such as operator overloading, are presently supported by CUDA and C++ is likely to become fully supported in forthcoming software releases.

We have developed a 3D Image Registration program, AIRWC (Accelerated Image Registration with CUDA), that is capable of performing very high resolution non-linear registrations in typically between four and eight minutes, corresponding to a 750 fold speedup on our conventional software running on a single CPU.

Reducing calculation times from days to minutes will enable numerous important applications including atlas building, and longitudinal studies of disease progression or normal aging. Direct clinical applications will include treatment planning for radiotherapy. We discuss the CUDA code and briefly present results from both human and rodent MRI imaging.

Manuscript received November 13, 2009.

R. E. Ansorge is with the University of Cambridge, Cavendish Laboratory Cambridge CB3 0HE UK (telephone: +44-1223-766103, e-mail: rea1@cam.ac.uk).

S. J. Sawiak., is with the University of Cambridge, Wolfson Brain Imaging Centre, Cambridge CB2 2QQ UK (telephone +44 1223 762128, e-mail sjs80@cam.ac.uk).

G. B. Williams, is with the University of Cambridge, Wolfson Brain Imaging Centre, Cambridge CB2 2QQ UK (telephone +44 1223 762128, e-mail gbw1000@wbic.cam.ac.uk).

II. GPU Architecture

Briefly the GTX 285 and Tesla C1060 GPUs used for this work consist of 240 processing cores arranged into 30 "multiprocessors" each having 8 cores. Separate threads run on each core, each thread executes the same code. GPU programs (called kernels) typically schedule many threads to run on each core, the GPU schedules interleaved execution of these threads to hide memory latency. For efficient use of the GPU, a kernel should use many thousands of threads and the threads need to address memory in a coherent way and minimize branching. Our GPU programs are typically written with either 128 or 256 executing threads per multiprocessor.

The GPUs have 1 or 4 GB of global memory which has high latency and small amounts of faster (cached) constant memory and shared memory local to each multiprocessor. The GPUs also support 3D texture memory which can be used for very fast trilinear interpolation. Heavy use of this feature contributes to the speed of our code. Full details of the architecture are given in [4].

III. Methods

Our code follows the NVIDIA CUDA programming paradigm whereby conventional C++ host code calls kernel functions which run parallel threads on the processing cores of the GPU. In practice we have found it rather straight forward to write the necessary code; indeed the code examples shown here could easily run on a single conventional CPU or MPI cluster with very little modification.

The AIRWC host code reads and writes Nifti format files, the input data type is converted to 32-bit float, which is convenient and is also the most efficient data type for the GPU. Native CUDA data types are used where possible, for example float3 is used for vectors. Dedicated texture lookup is used to perform the tri-linear interpolations needed when calculating cost functions.

AIRWC implements twelve parameter affine rigid registration followed by cubic B-Spline based non-linear registration. The B-Spline registration code starts with a 4x4x4 grid of control points which is successively refined by power of 2 subdivisions up to a grid having 131x131x131 points (corresponding to two voxel spacing for 256 x 256 x 256 images). To simplify indexing and grid subdivision the initial grid voxel spacing is rounded up to a power of two and the grid points are placed between voxels.

The affine registration is optimized using a simplex based search and takes a few seconds. The B-Spline registration is optimized using a gradient descent method. The calculation of

the necessary gradients with respect to up to 6,744,273 grid parameters is the most costly part of the calculation. In order to scale well with increasing grid subdivision the CUDA implementation uses either a complete kernel, single multiprocessor or a single thread to calculate one gradient depending on the number of gradients required. This approach means that the overall calculation times are only weakly dependant on the grid spacing. Typically we also use more iterations for the highest resolution grids.

We have implemented Normalized Correlation (NCORR) and Normalized Mutual Information (NMI) as cost functions in AIRWC [5]. The former is simpler to implement on the GPU but is only useful for intra-modality imaging. The latter requires a 2D histogram of source and target image values; we have used a fixed histogram size of 128x128 bins for the current work. At present we have not used any histogram smoothing, for example a Parzen window.

AIRWC was run on a DELL XPS 630i equipped with an NVIDIA GTX285 GPU and running Windows Vista. For comparison we used an in-house version of vtkCISG [6] running on a Linux system with 3.2 GHz Xeon processors. We have also tested our code on a recent DELL workstation equipped with a Tesla C1060 GPU and obtained identical numerical results although the Tesla card seems to run about 10% slower than the GTX 285.

IV. Kernel Code

Examples of our GPU kernels are shown in figures 1-4. Variable names starting with the letter 'c' refer to parameters stored in the constant memory of the GPU, these are preloaded by the host code. Figure 1 shows the aff_nmi_hist kernel which builds the 2D histogram required for the NMI cost function. The AtomicAdd function is used to serialise possible simultaneous updates of a particular histogram bin by multiple threads. If needed this is an expensive GPU operation and two significant optimizations are shown; firstly we use 16 parallel arrays which are subsequently reduced and secondly we do not explicitly add to the (0,0) bin which usually contains a great many entries. The kernel is called with sufficient threadblocks of size 16x16 threads to span an image x-y slice. Figure 2 shows the bs_ncorr_cost kernel which computes the normalized correlation cost function for affine plus B-Spline transformations.

```
01__global__ void aff_nmi_hist(float *target,int *hist)
02{
03  int ix = blockIdx.x*blockDim.x + threadIdx.x;
04  int iy = blockIdx.y*blockDim.y + threadIdx.y;

05  if(ix >= cvox.x) return;
06  if(iy >= cvox.y ) return;
07  int vsize = cvox.x*cvox.y*cvox.z;

08  float x = (float)ix;
09  float y = (float)iy;
10  float z = 0.0f;
11  float4 v =  make_float4(x,y,z,1.0f);
12  float4 r0 = make_float4(c_affmat[ 0],c_affmat[ 1],c_affmat[ 2],c_affmat[ 3]);
13  float4 r1 = make_float4(c_affmat[ 4],c_affmat[ 5],c_affmat[ 6],c_affmat[ 7]);
14  float4 r2 = make_float4(c_affmat[ 8],c_affmat[ 9],c_affmat[10],c_affmat[11]);
15  float sx =  dot(r0,v);
16  float sy =  dot(r1,v);
17  float sz =  dot(r2,v);
18  float sval = 0.0f;
19  int istart = iy*cvox.x+ix;
20  int zstep =   cvox.x*cvox.y;
21  int slice = HBins*HBins*(ix%HSlices);
22  int hx = 0;
23  int hy = 0;
    // fill histogram for nmi
24  for(int i=istart;i<vsize;i+=zstep){
25    sval = tex3D(tex1, sx, sy, sz);
26    v.z += 1.0f;
27    sx =  dot(r0,v);
28    sy =  dot(r1,v);
29    sz =  dot(r2,v);
30    hx =(int)( ((float)sval - cslow)*cswidth );
31    hy =(int)( ((float)target[i] - ctlow)*ctwidth );
32    if(hx>0 || hy>0)atomicAdd(&hist[slice+hy*HBins + hx],1);
33  }
34}
```

Fig. 1. The aff_nmi_hist kernel. This kernel calculates the 2D histogram of source and target voxel values required for the NMI cost function. The histogram dimensions are given by the defined variable `HBins` which is 128. The kernel is called with sufficient threadblocks of size 16x16 threads to span an image x-y slice. Lines 30-31 calculate the source and target bin coordinates and line 32 updates the histogram using atomicAdd. Two important optimizations are shown:

line 21 selects one of HSlices (typically 16) parallel histograms for this thread to use, thus reducing collisions between threads, and the if statement in line 31 prevents the (0,0) bin being updated, in practice (for our images) this bin contains many voxels. A subsequent kernel reduces the parallel histograms and infers the value of the (0,0) by summing the counts in all the other bins to find the "missing" counts.

```
00 __global__ void bs_ncorr_cost(float *b,float *t,float3 *d,float3 *cg)
01 {
02  int ix = blockIdx.x*blockDim.x + threadIdx.x;
03  int iy = blockIdx.y*blockDim.y + threadIdx.y;
04  if(ix >= cvox.x || iy >= cvox.y) return;

    //------------------ this for affine transformation -----------------------
05  float x = (float)ix; float y = (float)iy; float z = 0.0f;
06  float4 v =  make_float4(x,y,z,1.0f);
07  float4 r0 = make_float4(c_affmat[ 0],c_affmat[ 1],c_affmat[ 2],c_affmat[ 3]);
08  float4 r1 = make_float4(c_affmat[ 4],c_affmat[ 5],c_affmat[ 6],c_affmat[ 7]);
09  float4 r2 = make_float4(c_affmat[ 8],c_affmat[ 9],c_affmat[10],c_affmat[11]);

10  float3 s = make_float3(0.0f);
11  float target = 0.0f; float source = 0.0f; float sumst = 0.0f;
12  float sumss = 0.0f;    float fabcost = 0.0f;   // fabs cost function

13  v.z=0.0f;
14  int ivox = iy*cvox.x+ix;
15  int zstep = cvox.x*cvox.y;
    //-------------------- this for bsplines --------------------------------
16  float3 fill = make_float3(0.0f);

    // find bs control point to left of me
17  int bx = (ix+cedge.x)/cpdim.x + cguard.x;
18  int by = (iy+cedge.y)/cpdim.y + cguard.y;
19  int bz = (   cedge.z)/cpdim.z + cguard.z;   // iz=0 at first

    //  indicies in unifed bs lookup table (kx,ky,kz)
20  int kx = ((ix+cedge.x)%cpdim.x)*cstride.x + coffset.x;
21  int ky = ((iy+cedge.y)%cpdim.y)*cstride.y + coffset.y;
22  int kz = 0;
    //----------------- loop over z voxels for this x-y ---------------
23  for(int iz=0;iz<cvox.z;iz++) {
24    // affine transformation here
25    s.x =  dot(r0,v);
26    s.y =  dot(r1,v);
27    s.z =  dot(r2,v);
28    // this for bsplines
29    bz = (iz+cedge.z)/cpdim.z + cguard.z;
30    kz = ((iz+cedge.z)%cpdim.z)*cstride.z + coffset.z;
31    for(int jz=0;jz<4;jz++) for(int jy=0;jy<4;jy++) for(int jx=0;jx<4;jx++){
32      s += cg[cx2i(bx+jx-1,by+jy-1,bz+jz-1)]*cbs[kx*4+jx]*cbs[ky*4+jy]*cbs[kz*4+jz];
33    }
34    d[ivox] = s;    // save s - side effect for gradient calculation
35    target = t[ivox];
36    source = tex3D(tex1, s.x, s.y, s.z); // triliner interpolation
37    sumst += source*target;
38    sumss += source*source;
39    ivox += zstep;  // update slice pointer
40    v.z += 1.0f;
41  }
    //  all done, save sums for this thread
42  b[iy*cvox.x+ix]=corr;
43  b[iy*cvox.x+ix +zstep]=sumsq;
44 }
```

Fig. 2. The bs_ncorr_cost kernel, this computes the normalized correlation cost function for affine plus B-Spline transformations. It is called with sufficient threadblocks of size 16x16 to span an image x-y slice. Lines 2-5 determine a particular thread's x-y pixel value. Lines 7-9 store the current affine transformation parameters for subsequent use in lines 26-28. Lines 13-23 initialize working variables. The main loop 23-41 is over the z slices of the image. The 3-fold inner loop 31-33 calculates the B-Spline tensor product (in practice the innermost loop is unrolled). The indices bx-z point to the grid point to the left of the current voxel, kx-z point to appropriate B-Spline values in the lookup table `cbs`. The vector **cg** contains the current B-Spline displacements and cx2i is a helper function returning linear addresses. The vector array **d** in global device memory is used to store the current displacement **s**, these values are used by subsequent gradient calculation kernels. In line 35 the target voxel value is found the in array `t` (these values never change). The source image values are stored in the 3D texture `tex1` and in line 36 device supported trilinear interpolation is used for the transformed source voxel value. In lines 42-43 the final partial cost function contributions are stored in the device array **b**. The final summation over the elements of **b** is presently done on the host although this could be done on the GPU for a slight performance boost.

```
01 __global__ void bs_ncorr_grad(float *b,float *t,float3 *d)
02 {
03   int id = blockIdx.x*blockDim.x+threadIdx.x;  // ((bz*nz+by)*ny+bx)*3+tz
04   if(id  >= cgdim.x*cgdim.y*cgdim.z*3)return;
05   int tz = id%3;
06   id /= 3;
07   int gx = id%cgdim.x;
08   id /= cgdim.x;
09   int gy = id%cgdim.y;
10   int gz = id/cgdim.y;

11   int mxL = max(0,(gx-cguard.x-2)*cpdim.x-cedge.x);
12   int mxH = min(cvox.x-1,(gx-cguard.x+2)*cpdim.x-cedge.x-1);
13   int myL = max(0,(gy-cguard.y-2)*cpdim.y-cedge.y);
14   int myH = min(cvox.y-1,(gy-cguard.y+2)*cpdim.y-cedge.y-1);
15   int mzL = max(0,(gz-cguard.z-2)*cpdim.z-cedge.z);
16   int mzH = min(cvox.z-1,(gz-cguard.z+2)*cpdim.z-cedge.z-1);
17   if(mzH < 0 || myH < 0 || mxH < 0) return; // grid point has no influence
     // working variables
18   float tv0 = 0.0f; float sv0 = 0.0f; float sv1 = 0.0f; float sv2 = 0.0f;
19   float dss1 = 0.0f; float dss2 = 0.0f; float dst1 = 0.0f; float dst2 = 0.0f;
20   float3 s0 = make_float3(0.0f);
21   float3 s1 = make_float3(0.0f);
22   float3 s2 = make_float3(0.0f);
     // loop over voxels infuenced by this grid point
23   for(int iz=mzL;iz<=mzH;iz++){
24     int bz = (iz+cedge.z)/cpdim.z + cguard.z;
25     int jz = gz-bz+1;
26     int kz = ((iz+cedge.z)%cpdim.z)*cstride.z + coffset.z;
27     for(int iy=myL;iy<=myH;iy++){
28       int by = (iy+cedge.y)/cpdim.y + cguard.y;
29       int jy = gy-by+1;
30       int ky = ((iy+cedge.y)%cpdim.y)*cstride.y + coffset.y;
31       int ivox = (iz*cvox.y+iy)*cvox.x+mxL;          // first {x,y,z} voxel
32       for(int ix=mxL;ix<=mxH;ix++){
33         int bx = (ix+cedge.x)/cpdim.x + cguard.x;  // Lefthand g point {bx,by,bz}
34         int jx = gx-bx+1;                          // offset of g from LH b grid point
35         int kx = ((ix+cedge.x)%cpdim.x)*cstride.x + coffset.x;
36         s0 = s1 = s2 = d[ivox];
37         tv0 = t[ivox];
38         sv0 = tex3D(tex1, s0.x, s0.y, s0.z);
39         s1 += cgstep[tz]*cbs[kx*4+jx]*cbs[ky*4+jy]*cbs[kz*4+jz];
40         sv1 = tex3D(tex1, s1.x, s1.y, s1.z);
41         s2 -= cgstep[tz]*cbs[kx*4+jx]*cbs[ky*4+jy]*cbs[kz*4+jz];
42         sv2 = tex3D(tex1, s2.x, s2.y, s2.z);
43         dss1 += ((sv1+sv0)*(sv1-sv0)); // NB sv1*sv1-sv0*sv0;
44         dst1 += ((sv1-sv0)*tv0);       //sv1*tv0-sv0*tv0;
45         dss2 += ((sv2+sv0)*(sv2-sv0)); // NB sv2*sv2-sv0*sv0;
46         dst2 += ((sv2-sv0)*tv0);       //sv2*tv0-sv0*tv0;
47         ivox++;
48   } } }
  //----------------- done, store changes to cost function ---------------
49   int place = ((gz*cgdim.z+gy)*cgdim.y+gx)*3+tz;
50   place *= 4;
51   b[place  ] += dss1;
52   b[place+1] += dst1;
53   b[place+2] += dss2;
54   b[place+3] += dst2;
55 }
```

Fig. 3. The bs_ncorr_grad kernel. This kernel computes to gradient for the Normalized Correlation cost function with respect to displacement vector components. One thread calculates the gradient for one component of one displacement vector. The kernel is called with typically 192 threads per block and sufficient blocks to process the entire set of displacement vectors. In lines 3-10 the thread determines which component to process. Lines 11-16 determine the subset of voxels over which this B-Spline point has influence. Lines 23-47 loop over voxels and calculate *changes* in $\sum s_i s_i$ and $\sum s_i t_i$. Note the use of factorization in lines 43 and 45 to preserve accuracy using single precision. In lines 49-54: the final changes are stored in device memory for subsequent final processing on the host.

```
01 __global__ void bs_nmi_grad1(int *thist,int *shist,float3 *d,int *hist,int *hwork,int gstart)
02 {
03   int *change_2dhist = hwork + HBuffs*HBstep*blockIdx.x;
04   int *change_shist = change_2dhist + HBins*HBins;
05   int thread = threadIdx.x;
06   int gid = gstart+blockIdx.x;  // grid point for this thread
07   if( gid  >= cgdim.x*cgdim.y*cgdim.z)return;
08   int gx = gid%cgdim.x;
09   int gy = (gid/cgdim.x)%cgdim.y;
10   int gz = (gid/cgdim.x)/cgdim.y;
11   int mxL = max(0,(gx-cguard.x-2)*cpdim.x-cedge.x);
12   int mxH = min(cvox.x-1,(gx-cguard.x+2)*cpdim.x-cedge.x-1);
13   int myL = max(0,(gy-cguard.y-2)*cpdim.y-cedge.y);
14   int myH = min(cvox.y-1,(gy-cguard.y+2)*cpdim.y-cedge.y-1);
15   int mzL = max(0,(gz-cguard.z-2)*cpdim.z-cedge.z);
16   int mzH = min(cvox.z-1,(gz-cguard.z+2)*cpdim.z-cedge.z-1);
17   if( mzH<mzL || myH<myL || mxH<mxL ) return; //no influence, all thds exit

18   int mx = mxH-mxL+1; int my = myH-myL+1; int mz = mzH-mzL+1;
19   int ncount = mx*my*mz;
20   float3 s1 = make_float3(0.0f);
21   float3 s0 = make_float3(0.0f);

22   for(int t=thread;t<ncount;t+=HBins){ // voxels loop using 128 threads
23     int iz = t/(mx*my) + mzL;      // (ix,iy,iz) true voxel coords
24     int iy = (t/mx)%my + myL;
25     int ix = (t%mx)    + mxL;
26     int ivox = (iz*cvox.y+iy)*cvox.x+ix;
27     int hy = thist[ivox];
28     int bz = (iz+cedge.z)/cpdim.z + cguard.z;
29     int jz = gz-bz+1;
30     int kz = ((iz+cedge.z)%cpdim.z)*cstride.z + coffset.z;
31     int by = (iy+cedge.y)/cpdim.y + cguard.y;
32     int jy = gy-by+1;
33     int ky = ((iy+cedge.y)%cpdim.y)*cstride.y + coffset.y;
34     int bx = (ix+cedge.x)/cpdim.x + cguard.x; // Lefthand g point {bx,by,bz}
35     int jx = gx-bx+1;                         // offset from LH b grid point
36     int kx = ((ix+cedge.x)%cpdim.x)*cstride.x + coffset.x;
37     s0 = d[ivox];
38     int hx0 = shist[ivox];
39     for(int bxyz=0;bxyz<3;bxyz++){
40       s1 = s0 + cgstep[bxyz]*cbs[kx*4+jx]*cbs[ky*4+jy]*cbs[kz*4+jz];
41       float sv1 = tex3D(tex1, s1.x, s1.y, s1.z);
42       int hx1 =(int)( (sv1 - cslow)*cswidth );
43       if(hx1 != hx0 ) {
44         if(hy !=0 || hx1 != 0)atomicAdd(&change_2dhist[hy*HBins+hx1+HBstep*2*bxyz],1);
45         if(hy !=0 || hx0 != 0)atomicAdd(&change_2dhist[hy*HBins+hx0+HBstep*2*bxyz],-1);
46       }
47       s1 -= 2.0f*cgstep[bxyz]*cbs[kx*4+jx]*cbs[ky*4+jy]*cbs[kz*4+jz];
48       sv1 = tex3D(tex1, s1.x, s1.y, s1.z);
49       hx1 =(int)( (sv1 - cslow)*cswidth );
50       if(hx1 != hx0 ) {
51         if(hy !=0 || hx1 != 0)atomicAdd(&change_2dhist[hy*HBins+hx1+HBstep*(2*bxyz+1)],1);
52         if(hy !=0 || hx0 != 0)atomicAdd(&change_2dhist[hy*HBins+hx0+HBstep*(2*bxyz+1)],-1);
53         // all done, next reduce data in seperate kernel!
54 } } } }
```

Fig. 4:. The bs_nmi_grad1 kernel. This kernel helps compute the gradient of the NMI cost function with respect to components of the B-Spline displacement vectors. Each thread block processes one displacement vector and has HBins (typically 128) threads, thus one thread processes one row (or column) of the 2D histogram. In lines 3-4 pointers to buffers in global device memory for the 2D and 1D source voxel histograms are set for this block. In lines 6-10 the grid point to be processed is determined, and then lines 11-17 set the range of voxels over which it has influence. Lines 22-54 are main loop over voxels, this is similar to that in fig. 2 except that HBins threads share the calculation. The inner loop, Lines 39-53, processes all three components of a grid point. If a source voxel changes its histogram bin due to a step in a component of this B-Spline displacement vector, the histogram changes are accumulated in change_hist_2d. A subsequent kernel is used to compute the change in the NMI cost function itself using the already known histogram without displacements and the change_hist_2d displacements.

Figure 3 shows bs_ncorr_grad kernel which calculates the gradient of the current cost-function value with respect to. the B-Spline displacement vectors. This kernel processes up to 3.5 10^9 voxels thus its performance is critical. In this version of the kernel one thread calculates the gradient for one component of a single displacement vector, this was found to be the most efficient allocation of work for the highest density of grid points. At lower grid densities similar kernels are used but with the calculation handled by single multi-processors (having 8 cores) or by an entire kernel at the lowest densities. Finally figure 4 shows the bs_nmi_grad1 kernel which computes the gradients of the NMI cost function by first

finding a histogram of changed voxels. Additional kernels (not shown) then process the output from this kernel. The figure captions give further detail.

An important feature of our code is that only a single interpolation step from the source to target image is performed during the optimization process and at the end for the final transformation. Additionally optimizations are always performed using the full resolution of the source and target images.

V. Results

Table 1 shows a timing comparison between AIRWC and our in-house version of vtkCISG 2.0 for the registration of two 240x256x176 brain extracted structural MRI images. The two subjects were healthy males of ages 25 and 62.

It should be pointed out that this was a preliminary test and that vtkCSIG used the NMI cost function whereas AIRWC used the less expensive NCORR cost function. Moreover on inspection some of the algorithms used in our vktCSIG code appeared to be suboptimal, for example a gradient decent method was used by both programs, but vtkCSIG used fixed step lengths whereas AIRWC uses optimal step lengths for a quadratic minimum. Nevertheless the speedup is still dramatic. The quality of the registrations is illustrated in Fig 5.

Table 1 Comparison of AIRWC and vtkCSIG

Grid Spacing (voxels)	vtkCISG Xeon 3.2 MHz, time (mins)	AIRWC GTX285, time (secs)	Speedup
256	14.9	21.3	50
128	110.5	41.6	159
64	342.5	116.4	177
32	645.8	274.5	141
16	1078.8	379.5	171
8	2070.5	394.4	315
4	3986.0	381.8	624
2	4919.0	394.4	748

We have now further optimized AIRWC and importantly implemented the NMI cost function, some more recent results for the same registration problem are shown in Table 2. The first row of the Table 2 shows the times required to evaluate the affine only cost function and the number of evaluations required for the Simplex optimization to converge. Rows 2 to 9 show the times required for one step of the B-Spline gradient optimization which requires evaluation of the gradient of the cost function with respect to all B-Spline displacements and three cost function evaluations. The last three rows show the total times taken and final cost function values. The last two rows of the table were calculated after registration using the final registered images. The times shown in this table are in seconds except for the affine transformation row where the times are in ms.

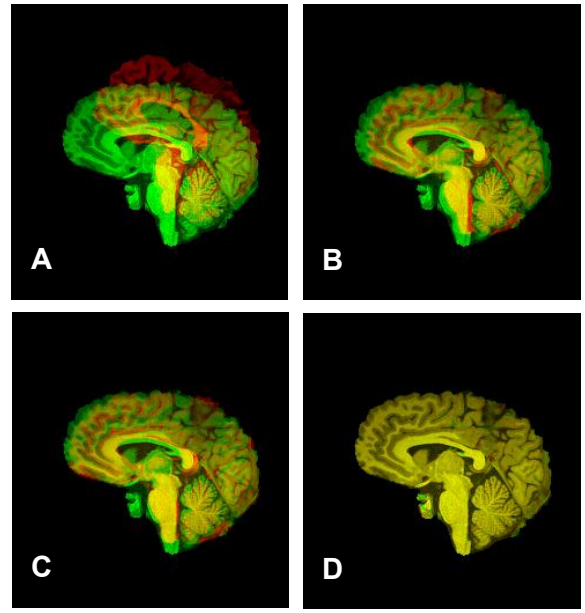

Fig. 5 Typical corresponding slices from inter subject registration, the two subjects are overlaid in read and green with matching regions shown in yellow. A: Un registered images, B: 6-parameter affine registered, C: 12 parameter affine registered and D: full B-Spline registration with one control point every two voxels.

Table 2 AIRWC timings.

Step	NCORR		NMI	
	time/ step	steps	time/ step	steps
Affine	3.88 (ms)	808	10.91 (ms)	1110
4x4x4	1.85	14	3.36	5
5x5x5	2.66	11	3.85	3
7x7x7	2.78	21	4.33	18
11x11x11	1.35	13	2.69	21
19x19x19	1.25	10	2.95	21
35x35x35	1.05	11	2.99	8
67x67x67	1.54	19	2.43	39
131x131x131	2.36	39	2.33	42
Total time	282.98		457.53	
NCORR	0.993015		0.992745	
NMI	1.39314		1.42336	

Figure 6 shows an "average MRI brain" derived from structural scans of 14 normal volunteers aged between 20 and 28. The average was computed iteratively. To start, an initial target image was made by affine registrations of 13 of these images to one chosen as typical and then averaging the results. All 14 images were then registered to this target using a 6-parameter affine transformation followed by 4x4x4 B-Spline registration. A new average was then made and used as the subsequent target. The B-Spline grid was iteratively refined to maximum resolution in seven steps. The final average represents some 112 3D registrations which in total took about 9.5 hours using AIRWC on the GTX825.

Figure 7 shows a registration between a preserved mouse brain and a similarly constructed atlas using 15 preserved brains; note the tear in the subject image has been essentially "closed" by the registration process. A deformation map using the Jacobian of the final transformation is shown in Fig. 6d and shows hot-spots of activity near distorted regions as expected. Such quantitative distortion maps are optionally output by AIRWC. In order to match the original source image, the Jacobian shown in Fig. 6d in fact corresponds to the inverse of the transform used to create the atlas, i.e. it is the transformation of the final atlas to the original source image. The speed of AIRWC makes the calculation of such inverses very practical.

VI. Conclusions

We believe that this application shows that GPUs have enormous potential for medical image registration. In particular a speedy way of generating precise local deformation maps between source and target images will be invaluable in many areas, including clinical studies of disease progression and deformation based morphometry applied to genetically modified rodent models.

Very fast affine registration is also potentially of great interest for near real time applications such as motion correction or tracking.

The code described here is available for non commercial use at http://www.bss.phy.cam.ac.uk/~rea1/AIRWC.html.

Acknowledgment

We acknowledge the help of colleagues at the Wolfson Brain Imaging Centre in providing our images.

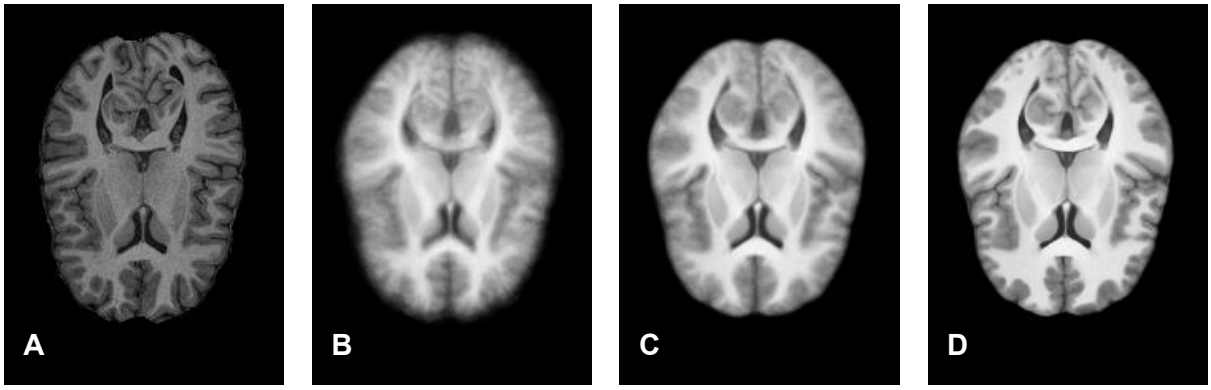

Fig. 6. Structural MRI Atlas built from 14 healthy volunteers using iterative method. A: single slice from one subject, B: Affine average after registration to one subject (rotation and translations only), C: iterative average after two B-Spline subdivisions, D: Slice from final atlas after seven subdivisions.

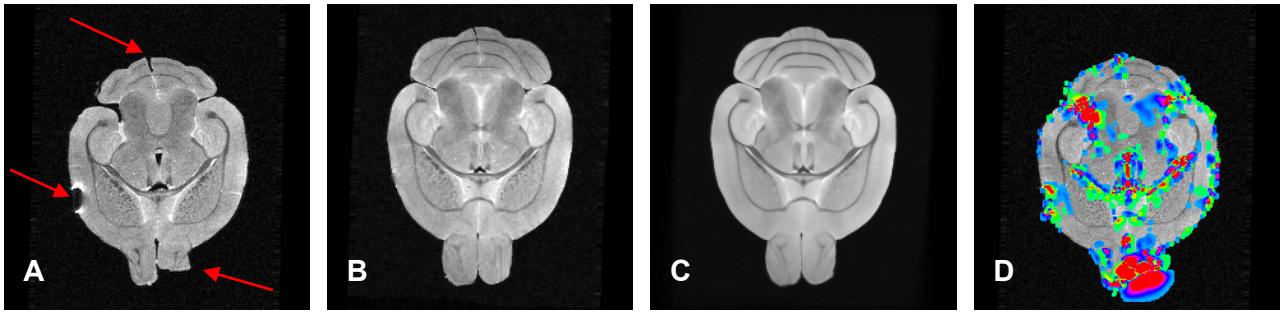

Fig.. 7. Structural MRI Atlas built from 15 preserved mouse brains using the same iterative method as figure 5. A: single slice from one subject showing damage from handling (red arrows). B: Same slice after final registration to atlas. C: Corresponding slice from atlas. D: Slice shown in A with Jacobian of transformation overlaid (red corresponding to the largest distortions).

References

[1] D. Blythe, *Proceedings of the IEEE,* 96 (2008) 761-778, "Rise of the Graphics Processor".

[2] F. Xu and K. Mueller, *IEEE Transactions on Nuclear Science*, 52 (2005) 654-663, "Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware".

[3] http://www.gpgpu.org/.

[4] http://www.nvidia.com/object/cuda_home.htm

[5] W. R. Crum, T. Hartkens, and D. L. G. Hill, *British Journal of Radiology*, 77 (2004) S140-S153 "Non-rigid image registration: theory and practice".

[6] C. Studholme, D.L. Hill, D.J Hawkes, *Pattern Recogn*. 32 (1999) 71-86, "An overlap invariant entropy measure of 3D medical image alignment".